Multi-Agent modelling, powered by ProbLog

Antonia Sewell

4th Year Project Report
Artificial Intelligence and Computer Science
School of Informatics
University of Edinburgh

2024

Abstract

Understanding the behaviour of autonomous entities, or ”agents,” in complex environments is a fundamental challenge with broad implications across various domains. This project aimed to address this challenge by developing Agent-Based Modelling (ABM) Software using ProbLog programs to model interactions among multiple agents. The contributions include the creation of a visual ABM software, the development of five agent-strategy programs in ProbLog and DTProbLog, and an analysis of their performance against various evaluation metrics. The report provides a detailed account of the project's background, strategy development, software implementation, experimental results, and discussion on limitations and future directions. Overall, this project contributes to the field of Multi-Agent Modelling by providing a flexible framework for simulating agent interactions and exploring strategies for optimising behaviour based on certain goals.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Antonia Sewell*)

Acknowledgements

I would like to express my gratitude to my supervisor, Vaishak Belle, for his unwavering support and guidance throughout this project. Additionally, I would like to thank my family for their constant encouragement, and my friends for the emotional support and fun times we share together.

Table of Contents

Chapter 1

Introduction

1.1 Motivations

We exist in a world full of autonomous entities, acting in the world making decisions based on factors around them. These entities could be humans moving around a supermarket, squirrels searching for food in a forest or companies seeking to make a profit. We can refer to these entities as 'agents'. What makes these agents act in the way they do? How do they make decisions? Are their decisions optimal, and if not what decisions might be? The investigation of the behaviours of agents in multi-agent scenarios is an interesting endeavour, one often approached by attempting to model their behaviour. This area is called Multi-Agent-Modelling.

Multi-Agent Modelling can be a challenge as the interaction of multiple agents is a complex problem, often with high computation needs in the modelling itself but also in the development of agent-behaviours using techniques such as reinforcement learning.

In attempting to model multi-agent behaviour and seeing to what extent the resultant behaviours correspond to or diverge from those in the real-world, insights can be gained into the possible logic behind them. And in experimenting with this logic, solutions can be found for real-world applications.

To investigate these models a general-purpose sandbox application within which rules for agents' behaviour can be defined and simulations run, could be an extremely useful tool. Some such applications, each with their own flavour, have already been created, which are described later in this paper.

To complement these it was decided to create an application to model multi-agent environments within which agents can move and consume various resources distributed in various locations within a two-dimensional environment, all obeying user-selected modes of behaviour. This could model anything from the behaviour of animals foraging for food, to minicabs collecting fares. But as well as mimicking behaviours in a 2D spatial context, simulations can be seen as modelling behaviours within more abstract conceptual environments; an example might be businesses competing for custom in a commercial environment.

The rules for defining the behaviours of the agents were written in the declarative logic programming languages ProbLog and DTProbLog, both derived from the declarative language Prolog. These languages were chosen for their succinct well-defined nature, their readability and their lower computing costs when compared with approaches such as reinforcement learning. These rules are referred to as 'Strategy-programs' throughout this paper and building these formed a large part of the project.

1.2 Contributions

The contributions of this project are:

1. The production of a visual Agent-Based-Modelling application, which allows users to specify the set-up of a world and observe how the agents interact with it.
2. The development of multiple agent-strategy programs in ProbLog and its variant DTProbLog.
3. Explanation and discussion of future strategies to be implemented.
4. An analysis of which strategies perform better against various evaluation metrics.

1.3 Report Structure

- **Chapter 2: Background and Literature Review** discusses concepts which are needed for understanding the remainder of the paper, as well as discussing the existing work in the field.
- **Chapter 3: The Problem** goes into detail about the Problem we attempt to solve, discussing what we mean by a 'goal', and details of allowed agent actions.
- **Chapter 4: Strategies** discusses the strategy-programs that were created in this project, giving insights into how they work, any challenges that arose, as well as some further ideas for strategies.
- **Chapter 5: Implementation of the ABM Software** discusses the design of the software that was created and the methodology of the implementation, and language decisions, are discussed in further detail.
- **Chapter 6: Results and Analysis** discusses the experiments performed to test strategies against various evaluation metrics.
- **Chapter 7: Conclusions** concludes on the findings of the project and discusses some recommendations for future work.

Chapter 2

Background and Literature Review

In this section the key concepts necessary for understanding the project are introduced. We start with decision theory and its relevance to agent-based systems, then move on to explore multi-agent decision theory. We'll also dive into DTProbLog and its precursor, ProbLog, discussing their roles in probabilistic logical reasoning within agent systems. Additionally, we'll review previous related work to situate our project within the broader context of ongoing research. By doing so, we aim to identify areas for innovation and highlight the unique contributions our project brings to the field.

2.1 Decision Theoretic Agents and Multi-Agent Decision Theory

Decision theory serves as the theoretical framework for making decisions based on probabilities of outcome states. At its core, it seeks to identify the optimal choice, often defined as maximizing the **expected utility** of the agent. Here utility denotes a predefined value signifying the desirability of an outcome for the agent. A **decision-theoretic agent** operates by selecting actions that promise the highest utility.

To illustrate the concept of utility and expected utility, let's consider the scenario of an agent tasked with selecting the most promising rock to search for food. Suppose we assign a utility of 5 to finding and consuming a food item. Now, imagine two rocks—analogous to decision options—with varying probabilities of harbouring food: 0.8 for red rocks and 0.3 for blue rocks. We can determine the optimal choice by calculating the expected utility, which involves multiplying the probability of an event by its associated utility.

$$ExpectedUtility = Probability \times Utility \tag{2.1}$$

For instance, the expected utility of inspecting the red rock would be $0.8 * 5 = 4$, while that of the blue rock would be $0.3 * 5 = 1.5$. As 4 is greater than 1.5, the red rock emerges as the superior option due to its higher expected utility. We'll delve deeper into

this example when discussing the syntax of ProbLog and DTProbLog in Sections 2.3.2 and 2.3.3.

Multi-Agent Decision Theory concerns scenarios where multiple agents coexist and make decisions within the same environment. This field bears close resemblance to **Game Theory**, which examines the interactions between self-interested agents, each striving to maximize their individual utility. In this project, we explore ideas in both competitive and collaborative decision-making.

2.2 Existing Techniques for decision making

2.2.1 Exhaustive Techniques

A straightforward technique to employ when performing decision making is an exhaustive one where all options for action are searched through and the best one (the one that gives rise to the greatest utility) is selected as the choice. Of course, these techniques have their limitations, particularly regarding time complexity. The exhaustive techniques can therefore be infeasible to use, particularly in a dynamic and real-life environment.

In this project, we wanted to create a piece of responsive agent-based-modelling software. Using exhaustive techniques for agent strategies may create a slow and unsatisfying user experience, and not allow results to be produced quickly enough, so were generally avoided in this project. Although, we do create and test one exhaustive technique, which is discussed in Section 4.4.

2.2.2 Markov Decision Processes (MDPs)

Markov Decision Processes are centred around states which agents are in, actions, and rewards [5]. The technique is used in order to find good universal policies for an agent in a stochastic environment. The process works by visiting a series of states and observing the expected rewards. Solving an MDP means controlling an agent so that it gains the maximum reward that it can. Solutions to MDPs are usually given as universal plans or policies due to the stochastic nature of the world, meaning that the results of actions are not certain, and there not necessarily being only one solution. The world-set-up that we choose to explore in this project is one with deterministic actions, which is in contrast to many settings where MDPs are used. Although MDPs can be a worthwhile approach for developing agent strategies, we don't use them here, as our goal instead was to produce well-defined rules in ProbLog.

2.2.3 Reinforcement Learning

Reinforcement learning is a common technique for decision-making in agent systems. In reinforcement learning, agents learn to make effective action choices through interaction with their environment, aiming to maximize cumulative rewards over time. There have been many attempts at using reinforcement learning for 2D-game-like scenarios

similar to our own such as in this 2013 paper [2] which uses reinforcement learning to train Ms. Pac-man.

Reinforcement learning algorithms, such as Q-learning [18] and Deep Q-Networks (DQN) [10], have been successfully applied to various real-world problems, including robotics, autonomous driving, and recommendation systems. These algorithms enable agents to learn optimal policies for decision-making in complex and uncertain environments.

However, reinforcement learning techniques for agents are not without their downsides. Reinforcement learning techniques can be very slow due to the repetitive training needed. This would be prohibitive for this project in the instance of changing the world set-up due to the whole model needing to be retrained again.

Primarily, reinforcement learning cannot produce well-defined logical rules, which is an essential element of what this project set out to do.

2.3 Probabilistic Logic Programming

Probabilistic Logic Programming is the blanket term for languages where facts can be annotated by probabilities. There are many languages in this family to be considered including but not limited to: ProbLog [8], PRISM [15], CP-logic [17]. Probabilistic Logic Programming Languages all follow the distribution semantics [14] which is a semantics combining symbolic computation and statistical modelling. In turn these languages share the key structure that from probabilistic choices and a logic program a distribution of possible worlds is produced. In this project, we use the probabilistic programming language ProbLog, which is an extension o the logical language Prolog, and its decision-theoretic extension DTProbLog.

2.3.1 Prolog

Prolog is a declarative programming language rooted in First-Order Logic (FOL). In Prolog, a program is defined by a set of facts and rules, which together specify relations and define the behaviour of the program. Computation in Prolog occurs through the process of querying the program with specific goals or queries.

An example Prolog program is provided below:

```
hasFood(R) :- rock(R), food(F), at(R, F).
```

This Prolog rule specifies that a rock R has food under it if there exists a food F that is located at rock R. In Prolog syntax, the `:-` operator can be read as ”if” or ”is true if”. Thus, the rule can be interpreted as ”hasFood(R) is true if there exists a rock R, a food F, and F is at R”.

To illustrate the use of this rule, suppose a query is made providing evidence that there exists food at rock r. The query `hasFood(r)` would then be posed to the Prolog program. If the evidence aligns with the rule, the query response would be true, indicating that there is indeed food under rock r.

Queries in Prolog follow a specific format. Typically, a query consists of a predicate followed by a list of arguments enclosed in parentheses. For example, to query whether there is food under rock *r*, one would pose the query `hasFood(r)`. Prolog evaluates this query by attempting to match it with the defined facts and rules in the program, ultimately returning true or false based on whether the query can be satisfied.

The format of a query in Prolog is as follows:

```
predicate(argument1, argument2, ..., argumentN).
```

Where `predicate` is the name of the predicate being queried, and `argument1`, `argument2`, ..., `argumentN` are the arguments passed to the predicate.

2.3.2 ProbLog

ProbLog extends Prolog by allowing users to assign probabilities to facts and rules, enabling reasoning under uncertainty. In ProbLog, a theory T consists of a set of probabilistic facts F and a set of clauses BK representing background knowledge.

A ground probabilistic fact (where all values have been replaced with specific values, resulting in a full instantiated fact) in ProbLog is denoted using the syntax: `p::f.` (including the "." symbol). `f` is a ground fact annotated with a probability `p`. ProbLog allows probabilistic clauses which are statements of the form `p::f:-body`, where the body is a conjunction of calls to non-probabilistic facts. This means that if `body` is true, then the probability of `f` is `p`. Probabilistic annotated disjunctions can also be expressed in the form

```
p_1 :: f_1 ;...; p_n :: f_n :- body
```

When queried, a probability distribution is defined over the two-valued well-founded models of the atoms in the program. The probability of a model is defined as

$$P(M) = \prod_{l \in M} P(l)$$

where the product runs over all the literals in the model M. For a query atom q the distribution semantics defines a probability for the query

$$P(q) = \sum_{M \models q} P(M) = \sum_{M \models q} \prod_{l \in M} P(l)$$

An example of a ProbLog program is depicted below. This program defines probabilistic facts about rocks having food under them based on their colours.

```
% Probabilistic facts
0.8 :: hasFood(R) :- rock(R), red(R).
0.3 :: hasFood(R) :- rock(R), blue(R).

% Background knowledge
rock(r1).
```

```
rock(r2).
red(r1).
blue(r2).

query(hasFood(r1)).
```

In this example, rocks $r1$ and $r2$ are defined, along with their colours. The program specifies that there is a 0.8 probability for a red rock having food under it, and a 0.3 probability for a blue rock having food under it. The query `query(hasFood(r1))` returns the probability 0.8, reflecting the likelihood of $r1$ having food based on the defined probabilities.

2.3.3 DTProbLog

The paper "DTProbLog: A Decision-Theoretic Probabilistic Prolog" [16] introduces a decision-theoretic extension to ProbLog. Using this framework, a decision is made by calculating expected utility values and choosing the choice with the highest one.

In contrast to standard ProbLog, in DTProbLog there are no queries, and instead, the program outputs the expected utility values for choices which the programmer gives. As well as the background knowledge BK and the facts with their probabilities, a DTProbLog program has a set of decision facts D and utility attributes U.

In DTProbLog, utilities are defined: `utility(eatFood(R), 5) :- rock(R).` This states that the utility of eating food from any rock R is equal to 5. An example of a DTProbLog program is shown below.

```
% Decisions
% whether or not to look under rock R
? :: lookUnder(R) :- rock(R).

% Utility attributes
utility(eatFood(R), 5) :- rock(R).
utility(lookUnder(R), -1) :- rock(R).

% Rules
eatFood(R) :- lookUnder(R), rock(R), hasFood(R).
```

When adding this code onto the code in the previous section (and removing the query), we have a DTProbLog program that is ready to be solved. When the program is solved, for each decision, 1 is outputted if making that decision results in a positive utility, and 0 in the case of a negative utility. Here, atoms `lookUnder(r1)` and `lookUnder(r2)` result in a positive utility, and `lookUnder(r3)` in a negative utility. Therefore actions `lookUnder(r1)` and `looKUnder(r2)` should be performed, but `lookUnder(r3)` should not.

2.4 Agent-Based Models (ABM)

Agent-based models involve the simulation of multiple autonomous agents, each operating independently within a shared environment.

There is a distinction to be made between ABMs and Multi-Agent-Systems (MAS). Unlike MASs, ABMs are more focused on simulating the actions and interactions of agents, rather than using multiple agents to solve a particular problem. Here, we focus on ABMs, as the project is interested in the ease of modelling multi-agent simulations, and not focused on solving a particular problem using multi-agents

2.5 Existing ABM Software

The following section explores already-existing ABM Software and compares them with what this project aims to produce.

2.5.1 PsychSim

PsychSim [12] is a social simulation tool developed to model the theory of mind. The referenced paper presents an example of a classroom containing agents with different mental models, including bullies whose goal is to increase power, and teachers whose goal is to minimize harm done to students.

In PsychSim, each agent maintains decision-theoretic mental models containing their beliefs about each other agent. The models in PsychSim involve finite nested beliefs meaning agent A may have a belief about what B believes about what A believes down a certain number of levels. The finite nesting structure used is similar to the one proposed in previous work on the recursive modelling of other agents for decision making [4]. PsychSim used 2-level nesting as it was found this was sufficient to produce the desired behaviour.

While PsychSim provides an environment focusing on modelling theory of mind in life-like scenarios, the aim of this project focuses less on specifically social scenarios and instead provides a framework suitable for a general selection of scenarios, that could be more survival-based.

2.5.2 AgentScript

AgentScript [11] is an open-source Javascript library for writing agent-based models. It provides an easy way to instantiate multiple agents which each act in the world in the same way. The example projects on their website include ant-pheromone simulations and a flocking simulator.

AgentScript provides an easy and flexible way of creating and visualising agents in a 2D world. As a Javascript library, the behaviour of the agents is by definition defined procedurally, in contrast to this project where behaviour is defined declaratively.

2.5.3 GAMA platform

GAMA Platform [7] is a recently developed platform for agent-based simulation built for use by domain experts to model scenarios without needing explicit knowledge of coding. The GAMA Platform website provides tutorials including traffic modelling and predator-prey models. Within the GAMA Platform, there is an inbuilt visual library which can be used to display and represent the models created, including 2D and 3D environments.

When considering the design of this project, GAMA platform was looked at. The visual environment and ease of use without need of coding were attractive features. To that end, this project provides a visual UI where the simulation can be run and interacted with without the need for coding.

2.6 Existing Agent Worlds

Here we look at two existing types of world set-up for agents that helped inform the choice of world set-up in this project.

Predator-Prey simulations model the interactions between predators and prey in a shared environment. In this 2006 article [3], both predators and prey are allowed to move N, S, E, W. Predators eat prey with some probability. This world set-up shares similarities to the world set up in this project, where agents can eat inanimate food objects in the world and move N, S, E, W. This Agent-Based model differs from this project in that the movements of all agents are random, and it's intended to provide a simplified example of the population dynamics, rather than an evaluation of agent-strategies. In the world set up in this project, there are no predators and only one type of agents. Although, in Section 7, we discuss potential future expansions that involve multiple types of agents. [ensure we do]

Pac-Man is a classic arcade game where the player controls the titular character, Pac-Man, navigating a maze while avoiding ghosts and collecting rewards (dots). The objective is to clear all dots from the maze while avoiding ghosts' capture. Pac-Man can also collect power pellets to temporarily turn the tables and eat ghosts. Pac-Man has used as a benchmark for reinforcement learning algorithms such as in the previously mentioned paper [2] that uses RL to train Ms. Pac-Man. It provides a dynamic environment with clear objectives, making it suitable for evaluating agent performance. Agents in Pac-Man can typically move in four directions (up, down, left, right) and perform additional actions such as eating pellets or using power-ups.

Chapter 3

The Problem

The problem we try to solve is as follows: there is a set of agents in a 2D environment we call the world. A world has a 'size' of n, where the world is n by n cells. As well as agents, there are other static objects in the world which each have *value*. A representation of the world is seen in Figure 3.1.

Agents can perform a limited number of actions: move up, down, left or right, and consume objects that they are directly next to. At each time step an agent may perform 1 action. Agents have limited *health*. Each time-step, every agent's health is decreased by 1. When an agent consumes an item, they gain health of the value of the object. If an agent's health reaches 0, they are removed from the world.

The action an agent takes at each time-step is defined by the world's *strategy*. Each strategy is defined in ProbLog or DTProbLog. A key part of this problem and the main aspect that we are investigating is how the strategy employed by agents in the world affects how the simulation plays out.

It was decided that a grid layout would be used instead of a continuous layout to simplify the visuals and the process. We will discuss later (Section 5.4) why the choice of world specifications are not extremely important, as item-based strategies could be easily ported to a 3-dimensional or continuous world. It was decided that only one item would be allowed in each cell for easy viewing by the user. As well as this, the discrete grid representation allowed the implementation of cell-based strategies, which is a subset of the strategies implemented and will be discussed in section 4.4.

3.1 Agents

Agents have a position in the world as all objects do, and can perform one action per move. Agents can:

1. MOVE 1 SPACE
2. CONSUME 1 ITEM
3. DO NOTHING

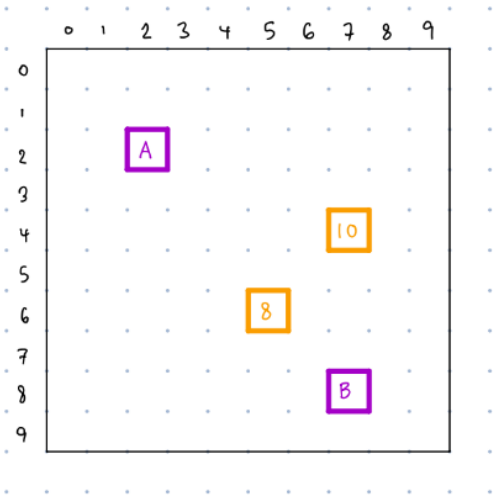

Figure 3.1: A representation of a simple world in our simulation context. Here the world size n is equal to 10 as the grid is 10 by 10. Boxes labels A and B represent two agents in the world. The other boxes represent objects with value values 8 and 10.

In each case, the agent can move to or consume an object in the cell directly up, down, left or right of it. An agent cannot move and consume an object in the same turn.

The 'do nothing' option is added for the case when an agent has decided that there is no point in moving. This happens in the ConsideringOthers strategy (to be discussed) in the instance where all items are being pursued by agents closer to the items than the acting agent.

These actions were decided upon as they allow the agents to move in the world in a simple and well defined way. Although we have defined the actions as above, it will become clearer further on in the paper (Section 5.4) that the definition of the actions is not of extreme importance, as many of the strategies (the item-based ones) have the ability be ported over quite easily to different world and action domains.

3.2 Goals

In this section, we discuss the concept of goals. We acknowledge the existence of multiple reasonable goals within our problem setting and explore the distinctions between collaborative and competitive goal scenarios.

A goal, in our context, can be loosely defined as the overarching objective that an agent or group of agents strives to achieve. It should be emphasised that our system does not explicitly define the goal for the simulation. Instead, the strategies are designed with implicit goals in mind, which likely represent various real-life scenarios.

These implicit goals may mirror scenarios encountered in domains such as restaurant-waiters, animals searching for food, and humans navigating a shopping mall, looking for items. Additionally, this approach contrasts with techniques such as Belief-Desire-

Intention (BDI) models of agents, which explicitly define the desires or motivational states of agents and derive actions accordingly [13].

In the subsequent sections, we delve into the implicit strategies that agents in our simulation may pursue to achieve their goals. These goals serve as the foundation for the evaluation (Section 6) of which strategies are most effective under different goal conditions.

3.2.1 The goal of an agent

In this subsection, we examine the diverse range of goals that an agent may pursue within the simulation environment. By considering examples and real-life agent scenarios, we aim to identify key objectives that shape agent behaviour and decision-making. These goals serve as criteria for evaluating the effectiveness of different strategies in achieving desired outcomes.

The goals of an agent can vary significantly depending on the context and objectives of the simulation. Some common examples include:

- **To survive for the longest time possible**
- **To survive longer than anyone else**
- **To eat as much value as possible in a lifetime**
- **To eat as much value as possible within the next K Moves**

Real-life goals could include **financial markets**, where agents may seek to maximize profits, minimise losses. In **natural ecosystems**, predators aim to capture prey for sustenance, while prey strive to evade predators and ensure survival. In the domain of **service robotics**, a waiter robot's goal may be to efficiently deliver orders to customers, optimize table turnover, and enhance the overall dining experience.

3.2.2 Sub-goal

We should also take some time to discuss the concept of a sub-goal. By sub-goal we mean the small goals that together (hopefully) form the completion of the wider goal. Examples of sub-goals that are encountered in this project include the next cell an agent should go to, or the next item that an agent goes for.

In determining agent behaviours, it's crucial to differentiate between the sub-goal of reaching a specific object and the sub-goal of reaching a particular cell within the environment. While these two objectives are interconnected, strategies within the simulation operate at different levels of abstraction, each with their own considerations and implications.

At the sub-goal level of reaching an object, agents aim to achieve specific objectives such as consuming items dispersed throughout the environment. The strategy program orchestrates this process by determining which item the agent should prioritize as its immediate goal item.

Conversely, the sub-goal of reaching a cell represents a more granular level of navigation, focusing on the physical movement of agents within the environment. In this project, this aspect is handled within the Python simulation program for most strategies (which is discussed further on in Section 5), where agents calculate the optimal path to reach their designated goal-cell. The decision for the simulation program to handle this was influenced by the open nature of the world, where obstacles are minimal, and route-finding complexity is reduced. However, one strategy diverges from this convention by operating at the cell-level, as discussed in further detail in Section 4.4.

3.2.3 The goal of a world

In addition to considering the goals of individual agents, we can consider the notional objectives of the world. In this context, the collective actions of all agents within the world may contribute to achieving a shared goal. This notion sheds light on the interconnected dynamics of the simulation environment and underscores the significance of collaborative or cooperative behaviours among agents. Some examples of potential world-goals include:

- **For all objects to be consumed as quickly as possible**
- **For the average lifespan of the agents to be as long as possible**
- **For high-value items to be consumed more quickly than low-value items**
- **For as much value to be consumed from the world in a time-span as possible**

The concept of the world having its own goals ties into the broader discussion of competitive versus collaborative strategies. Here, the goal of the world can be viewed as the collaborative objective shared by all agents inhabiting the environment. This concept will be explored further in Section 4.1, where we'll look into the implications of these dynamics using a robot-waiter example as a case study.

Furthermore, these world goals serve as the basis for evaluating the success of the strategies employed by agents within the simulation. In Chapter 6, we'll examine how these goals inform the evaluation metrics used to assess the effectiveness of different strategies in achieving desired outcomes.

3.3 Probabilistic Objects

The application has an option for probabilistic objects to be in the world. **Probabilistic objects** are objects within the simulation that have a certain probability of having a certain value. Objects in the world display a labelled value. We use the term 'real' to discuss objects the value of which is actually equal to their labelled value. Agents can perceive these objects in the world and know their labelled values, and the probability of them being real.

If the option for probabilistic objects is on (we discuss user options further in Section 5.6), a random number generator decides whether or not that object will be 'real', when it is added to the world. The object's value is displayed in black text if real, and red

text if not. The agents do not know whether an object is really real, all they know is the static probability of the item being real, and the labelled value. An expected utility calculation is then carried out considering this.

For example, there may be a probabilistic item in the world with a probability of 0.8. This means that any time the item of that type is put into a world, it has a 0.8 chance of being real. Let's say the value of the object is 10. It makes sense for the agent to perform a calculation to find the expected utility. The expected utility will then be equal to $0.8 * 10 = 8$. This calculation is performed when the agent is deciding their next goal item through the Strategy-program. The item with the highest expected utility will be the best choice in the instance where there are no other agents in the world.

3.3.1 Real-life probabilistic objects

Probabilistic objects exist in real-world applications such as computer vision where objects are classified with some certainty. Examples include the implementation from this 2020 paper [6] on Probabilistic Object Detection, which has a special emphasis on providing accurate probabilities on the certainty of classification. Autonomous vehicles often use object detection which provides some certainty score. (for further insight check [1] which discusses the status of this field and challenges). We could imagine that the system created in this project could be used to model robotic agents which are certain of the classification of an object to some degree of probability, which affects what the best decision for the agent is.

Chapter 4

Strategies

A significant part of the project concerns the strategies implemented in the simulation. The word 'strategy' refers to the method that the agents in the simulation employ when they are in the world. In this project, each strategy has a corresponding ProbLog, or DTProbLog program, which defines the agent's behaviour in different set-ups of the world. The strategies provide rules for the agents' behaviour.

4.1 Competitive vs Collaborative

A key idea that arose when thinking about strategies was the distinction between a competitive and collaborative environment and the potential differences this may have on strategies developed for achieving the goals. Both competitive and collaborative goals were considered in this project. In the following sections, we examine the distinction between a competitive and collaborative setting with an example involving waiter robots.

4.1.1 Competitive Goals and Strategies

Imagine a scenario where there are several waiter-robots (agents) in a restaurant who can pick up customers' orders (objects). First, imagine a competitive scenario where waiters want to pick up as many orders as they, individually, possibly can. Orders have different weightings (values) assigned to them, say due to priority. Agents are aware that collecting those orders of higher weighting is more rewarding, and should therefore attempt to gain as much reward as they can. They know that this is the goal of each of the other agents too. For the agent to maximise the number of orders it picks up, it will need to consider what the other agents will do in their positions.

It makes sense for agent A, who is closest to the order which has the most utility for them - considering the value of the item and the number of time steps if would take them to go there - to go to that order, as this is the best outcome for them and they know with certainty it can be achieved. This scenario can be seen in Figure 4.1 (here we remove agent B for simplicity of explanation). The agent who is closest to the order

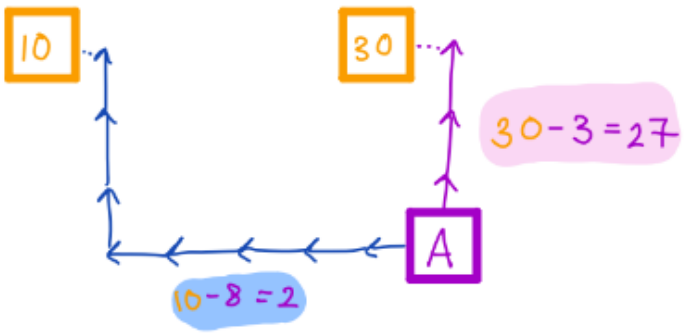

Figure 4.1: A depiction of an agent A considering its two options for goal item. For the item labelled 30 (meaning the item has a value of 30, the utility is calculated at the value minus the number of moves it takes agent A to get there (3). The same is done for the item labelled 10. As $27 > 2$, the item labelled 30 is the better choice.

that is best for them knows that even if every other agent started going there, it would arrive first. For now, we assume that once every agent reaches an item it stops moving.

Now imagine another agent, B, who is the second closest to the best order for them. B will need to consider whether agent A who is closer to that order will choose that item to go to. Agent A will choose to go there (if they are using the ConsideringOthers strategy, which we explore in more detail in Section 4.3) unless there is a better order for them closer. This scenario can be seen in Figure 4.2

If the closest agent, A, will not end up going there due to having a better option, then B can go there without worrying about A arriving there first. But if it turns out that A's best option is to go to that item, then B should not go there, as A will definitely arrive first, leaving B with nothing. B should then consider their next best option.

4.1.2 Collaborative Goals and Strategies

Now let's imagine a similar scenario also involving waiter-robots in a restaurant. Conversely to our previous waiter example, let us imagine the goal is that all of the customers are served as quickly as possible. This is different to the previous example as this is a collaborative goal amongst the agents in the world.

Intuitively one might think that the behaviour for the competitive and collaborative scenarios we have just discussed would result in similar behaviour, however there is a distinction. For example, it is seen in Figure 4.3 that the behaviour which would minimise the time taken for all customers to be served differs from what the agents would do if they were both self-interested and trying to maximize their own personal utility. The strategy that would best suit the scenario when the speed at which all customers are served (or objects are consumed) is discussed in Section **??**. We discuss the ConsideringOthers strategy further in Section 4.3.

The following sections describe the strategies that were implemented, how they work,

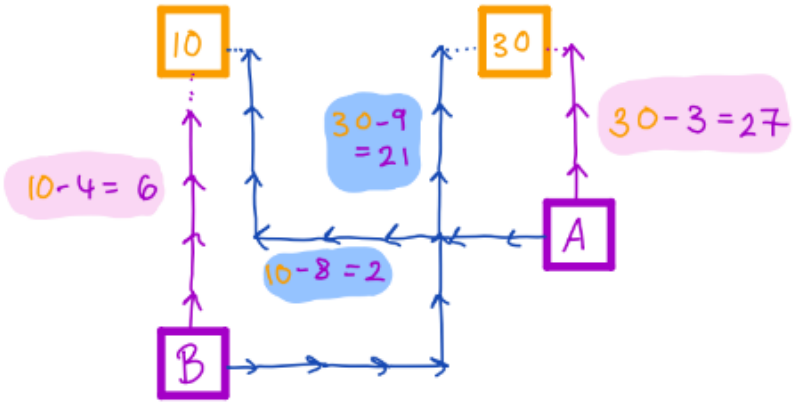

Figure 4.2: A depiction showing two agents, A and B, each considering their options for goal item using the ConsideringOthers strategy. The decision ultimately decided on is highlighted in pink. A chooses its most utilitarian option which would reward them with 27 value if they got it. This is due to A being certain it can get there without any other agent getting there first (A is the closest to its favourite item). B considers this and decides to go for the item labelled 10, even though its the worse option, as it knows A will get to the best one first.

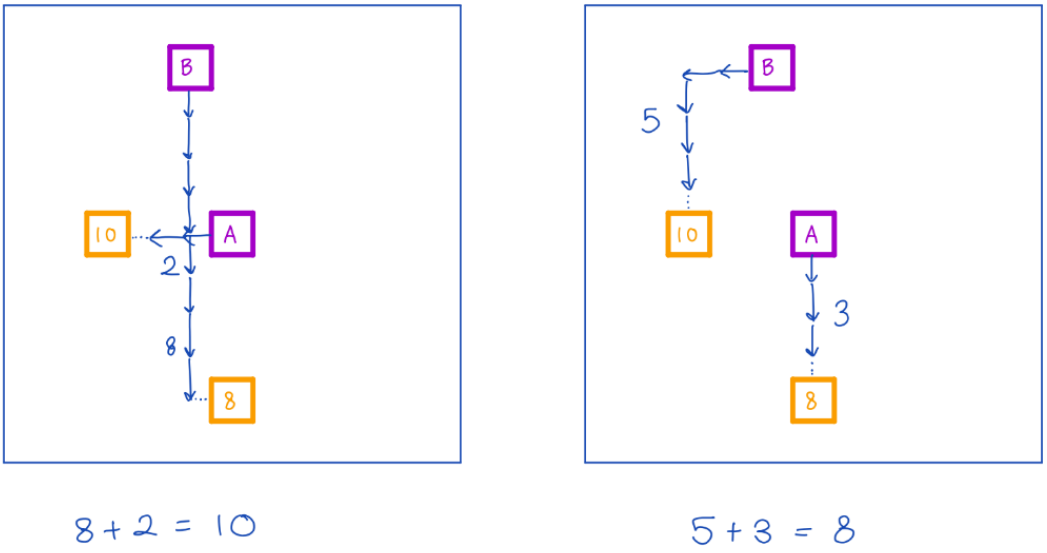

Figure 4.3: Two illustrations showing two possible outcomes for agent actions in the same world scenario, depending on the strategy implemented. Left, we see the competitive ConsideringOthers strategy in place, with A taking its best option, and B taking this into consideration and going for the next best option. Right, we see a collaborative strategy in place, where the world goal is for all objects to be consumed with as little distance being travelled as possible. Therefore agents A and B go for items as shown, minimizing the total distance travelled, equal to 8, which is less than 10

and why it was decided for them to be implemented in that way.

4.2 Greedy Strategies

This section describes three strategies referred to as 'greedy' strategies which work by going to the items they consider 'best' without considering any other factors. The following strategies consider different combinations of factors to be most important when deciding on the next goal item.

4.2.1 Greedy-No-Distance Strategy

Initially, a naive implementation of a Value-only Greedy strategy was implemented that ignores the distance of items and prioritises higher-value items, no matter how far they may be from the agent. The utility rule is seen below:

```
utility(A, F, X) :- food(F), agent(A), value(F, X).
```

This states that the utility for agent A eating food F is equal to the value of food F.

It should be noted that the result of this strategy is that every agent in the world is pursuing the same, highest-value item. This results in it taking longer for many items to be consumed over time.

4.2.2 Greedy-No-Value Strategy

The next strategy that was implemented instead considered only the distance the agent is from the items. Only a very subtle adjustment needed to be made in the Strategy rules code which was that instead of value, cost (equal to the negative distance) was set as the utility instead.

```
utility(A, F, C) :- food(F), agent(A), cost(A, F, C).
```

This strategy may be useful in a scenario where we want to switch into a mode that ignores the value of items. In the waiter example, this would mean treating all customers as equal without giving them any priority.

4.2.3 Standard Greedy Strategy

Combining the two previous strategies, in the standard greedy strategy, agents move towards the most valuable food for them considering the overall utility it would bring to them. This is equal to the value of the item minus the number of time steps it would take for them to arrive there. This can be expressed using the equation below:

$$U(O) = V(O) - T(O) \tag{4.1}$$

Where U is the utility of the object, V is the value of the object, and T is the number of time steps it would take for the agent to get there. The agent should pick the item with the highest utility, this can be expressed with the equation below:

$$G(O) = argmax_O\,(V(O) - T(O)) \tag{4.2}$$

Using this logic, we define the utility attribute in the Strategy program as:

```
utility(A, F, X + C) :-
    food(F), agent(A), value(F, X), cost(A, F, C).
```

In this strategy, the cost predicate is defined as:

```
cost(A, F, -X) :- distance(A, F, X), food(F), agent(A).
```

Where X holds the number of time steps it would take for agent A to get to food F.

4.2.4 Probabilistic Greedy Strategy

Probabilistic variants of the strategies were added which allow probabilistic objects in the world as discussed in Section 3.3. These strategies define utilities for items to use the expected utility equation instead of the value minus the cost. This change can be seen in the code below.

```
expected_utility(A, F, P * S) :-
    utility(A, F, S),
    probability(F, P).
```

Opposed to the previous section of code for the standard greedy strategy, this one multiplies the utility by the probability to get the expected utility, such as expressed in Equation 2.1.

Although ProbLog has an inbuilt system for handling probabilities, a decision was made in this strategy to define a custom predicate for dealing with them. This was mainly because there was difficulty with using the inbuilt probabilities with the utilities. To calculate the expected utility, the value of the objects had to be multiplied by the probability of the object, but there was no way of doing that when using the probability tools of ProbLog. This challenge is discussed further in Section 5.2.3.1.

4.3 ConsideringOthers Strategy

In this strategy, each agent calculates their own best move by considering the best moves for other agents in the world.

The strategy works for agent A by first calculating each agent's 'favourite' item, which is the one that would bring them the most utility if they existed in the world alone. Let's refer to A's favourite item as F. When considering whether it would be possible for A to get F, only agents which share it as their favourite item are considered. If there exists an agent B, for whom F is also their favourite, that is closer than A then it no longer makes sense for A to consider F, as it is certain that another agent will reach it before them.

The strategy works by iteratively filtering out all agents that are closest to their favourite item, along with that item. After each iteration we are left with a similar problem but

with less items and less agents to consider. This happens until the agent closest to their favourite item is agent A. Then agent A can set the item as their goal item knowing that it is its best option if we consider an environment where any agent only gets one item and then stops searching.

Care had to be taken when expressing this in ProbLog to avoid an infinite cycle error. For this reason list rules were added to the program. This is discussed further in Section 5.2.3.1.

A problem with the above approach is its computational load. There is repetition in the calculations of the goal items for agents, due to agents performing calculations of many of the other agents' goal items. A communication setting was added to the application to combat this. This allows the computation of which item each agent's goal item to be calculated only once per time step, instead of as many times as there are agents in each time step. This increases the smoothness of the simulation run.

This setting changes the behaviour in the agents ever so slightly due to the fact that normally the calculation is made after every agent moves in order. For example, in ConsideringOthers without communication Agent 1 will calculate their move, move, and then Agent 2 will calculate their move having seen the outcome of Agent 1's move. But when communication is on, every agent's move is calculated from the same exact state which is at the beginning of the time step. This also causes unnecessary agent movements when two agents are equidistant from the same item.

4.3.1 Strategy Discussion

As the agents exist in a deterministic world, where actions are certain to be successful, and it is known that all agents use the same logic to decide on the best item, they can work out which the optimal goal item for them would be by considering what the best options are for the other agents.

This strategy would perform optimally in a world where agents stop moving and consuming objects once they have consumed their first one. This could be thought of as similar to a scenario where agents take time to consume an object. For example, in the waiter-robot example, it could take time for a robot to pick up an order. In that case, it would become less likely that an agent could consume two objects in the time another could reach and consume one.

However, in the scenario where agents can consume objects and then carry on consuming objects immediately afterwards, this strategy is sub-optimal as there may be an instance where an agent A can reach the best item and another item in the time it takes for another agent B to get to the second one as in 4.4. This would have been foreseeable if it was considered what agent A's best option after consuming its first item would be. This idea links to the K-moves ahead strategy idea which is described in Section 4.5.1.

It is interesting to note that in the case where agents stop moving once they've consumed an item, this strategy achieves Nash Equilibrium, a concept in Game Theory which describes a state in a game where no one player can gain anything by changing only their own strategy.

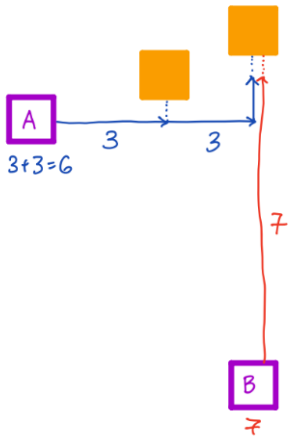

Figure 4.4: A depiction of a scenario where agent A can reach both its initial goal item, F_1, and its second goal item F_2 in less time than agent B can reach F_2.

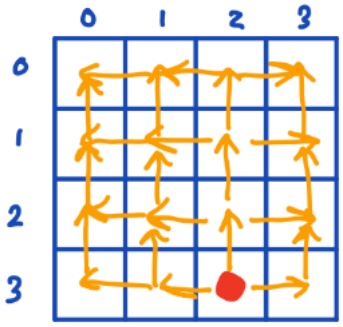

Figure 4.5: A diagram depicting an agent A (in red) and the cell connections between cells that are generated by `problog_maker`. The connections point outwards, reducing the number of potential routes to search through.

4.4 ProbLog Cell Strategy

This strategy represents interconnected cells and querying the program as to how to maximise utility. The purpose of this strategy was to test the viability of implementing a strategy in this way. Due to this being an exhaustive method, it is very slow and computationally complex and this complexity means it is not viable to use with multiple agents and also greatly limits the size of the grid.

A design feature of the ProbLog Cell Strategy is the use of single-directional cell connections which are demonstrated in Figure 4.5. This design choice not only prevents infinite cycles but also increases efficiency by guaranteeing the fastest route between cells based on the problem setup. In order to get the single-directional cell connections, an algorithm was implemented that used a method of going out from the agent in diamond shapes 4.6 through using breadth-first recursion. This ensured that the connections were generated in the correct order.

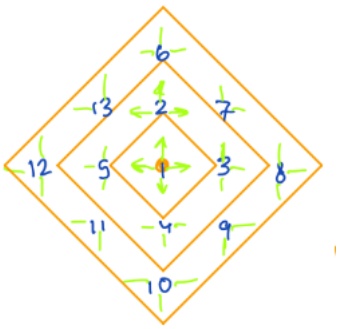

Figure 4.6: A drawing showing the order in which cell connections were generated, starting from the middle cells and going outwards in diamond shapes. At each cell, where connections were not present, connections were made with adjacent cells. This ensured that the connections were added in the correct order.

4.5 Further Strategy Ideas

This section discusses potential further strategy ideas which were not implemented in this project but came up for consideration. It discusses how these could be implemented and some thoughts on how they may perform.

4.5.1 K-Moves ahead strategy

As previously discussed, in the ConsideringOthers strategy there are instances where an agent can consume a second item in the time it would take for another agent to reach that item, causing unnecessary movement for the other agent (as in Figure 4.4). This strategy solves this problem by building on the ConsideringOthers strategy so that multiple objects are considered instead of just the next one. This would mean that instead of an agent just considering what others' next items should be, they also consider what the ones after that should be.

Let's explain the strategy first by setting K=2. The strategy could work by calculating each other agent's first item they would go for F_1 would be and then what their next item would be after they've eaten their first, F_2. If an agent A can get the item they are about to go for (F_1) and reach an agent B's favourite item (F_2) in the time that agent B can reach it, then there is no point in B going for F_2. This would be an improvement on the current ConsideringOthers strategy, where B would waste its time attempting to reach the item, but only stop and pursue another item once A consumed its first item, realising that A will reach it before B does.

As K increases in size, it is expected that the accuracy of the strategy will increase, with agents spending less time pursuing items that they do not ultimately get. However, with this, the time complexity is sure to dramatically expand.

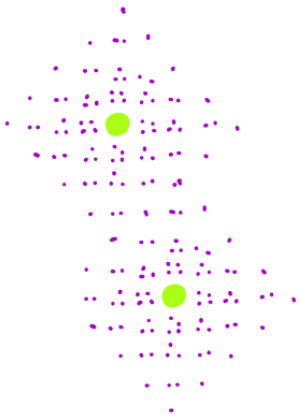

Figure 4.7: A depiction of the weighting placement for the Gradient Ascent strategy idea. The items are shown in green. More dots indicate a higher weighting. The weightings would decrease as they got further away from the cells containing items.

4.5.2 Gradient-Ascent strategy

The idea for this strategy is cell-based and uses a different technique than the others. It was inspired by techniques such as Ant Colony Optimisation (ACO) which uses position weightings to guide agents to a goal. In ACO, pheromone is placed in a world in order to optimise a route. In contrast, the 'pheromone' used here is set, based on the location of objects in the world. Objects with higher values will emit weight proportional to their value, getting less and less as cells go further away from them. The idea is that better cells for the agents to go to (with more high value objects) will have higher weight. The agents can traverse the map by 'climbing' the gradient, aka. going to the cell next to them with the highest weight and continuing to do that until they reach an item. Figure 4.7 shows a representation of the values that would be placed on cells, with more dots meaning a higher value placed on that cell.

4.6 Discussion

Although the strategies were implemented with a spatial 2D world in mind, they are applicable in scenarios even where there's no spatial world. The item strategies revolve around deciding which item to attempt to get, considering the fact that there are other agents also trying to gain the most utility they can.

One example of a real-world scenario like this might be companies deciding whether to bid for different contracts. Each would have to decide whether the investment in time and cost in preparing a pitch would be worth it bearing in mind the likelihood of better-placed competitors also bidding. It could be argued that the optimum strategy for this would be almost identical to the ConsideringOthers strategy in that those companies less well-placed would have to make a greater investment to secure the contract which would be wasted if they knew that a better-placed company is also likely to bid and win the contract.

Chapter 5

Implementation of the ABM Software

In this section, we discuss the design of the elements of the software system including the choice of language for different components, the class relations for the Python code base, details of classes, strategies, the user interface, and ProbLogMaker - a class made for the custom generation of ProbLog and DTProbLog programs.

5.1 Design Choices

This section discusses some of the decisions that were made when developing the Agent Modelling Software. When making design decisions, other existing platforms were taken into consideration as mentioned in Section 2.5.

It was an important criterion that the user could clearly understand the world's layout, the importance of which was reinforced when looking into existing ABM software such as AgentScript and GAMA Platform, each of which have inbuilt visual representations of the agent environments. A simple grid representation was decided on, which represents agents and items as squares of different colours, this is in contrast to GAMA Platform that has a 3-dimensional visual representation of agents, and AgentScript which has a continuous 2D visualisation of the agents.

There are different types of items each with their own value and colour. Each type's colour is generated automatically by the software along with a colour key. Agents are consistently represented by the same colour with white writing representing how much remaining health they have, which is increased by the value of consumed objects and decreased by 1 at each time step. The items each have a number representing their value.

It was an important criterion to allow the user a level of freedom and control when it came to the running of the simulation and to this end several buttons, sliders and selectors were added so that the user could change options from within the software. An image of the buttons and sliders can be seen in A.1.

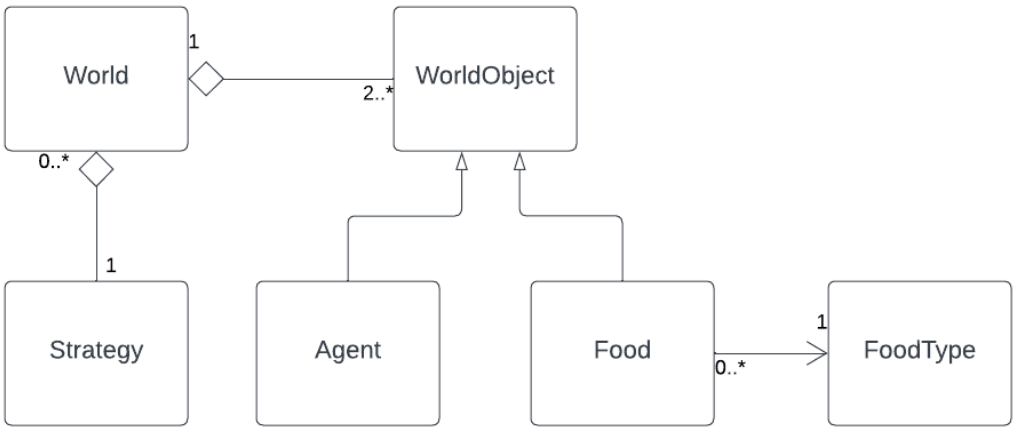

Figure 5.1: A class diagram depicting the classes in the simulation package and how they relate to one another, in UML form.

5.2 Language

Python is used as a front-end language which interfaces with strategies written in ProbLog. In this section we discuss the languages used, what they were used for, why it was decided that they were to be used, and challenges faced in using the language. Object Oriented Programming in Python was used to define classes for each part of the simulation as well as for the strategies. A class diagram can be seen in the 5.1.

5.2.1 Python

Since GUIs are not easy to create using Prolog, a high-level language was used for this purpose, as well as for designing the mechanics of the simulation. The language of choice was Python, due to its easy interfacing with ProbLog. ProbLog is written in Python and has an easily accessible Python API which made it straightforward to surround the ProbLog strategies and connect them to the simulation.

The Python package **Tkinter** was used to create the visual interface for the simulation. Tkinter provides tools for creating a simple window-based GUI. For more details on the Tkinter package see [9].

5.2.2 Potential benefits of using ProbLog and DTProbLog for writing the strategies

There are several reasons why defining the strategies in ProbLog rather than in Python or another procedural language was chosen. The main reason was in these languages strategies are expressed as logical constructs by which a decision is arrived at as opposed to procedures to be followed, which perhaps more closely resembles intellectual decision-making in the real world.

Another benefit of using ProbLog is that it is a declarative language, allowing strategy

writers to worry less about how the computation is calculated behind the scenes.

Also, defining the strategies in a different language than the one that the simulation was run in ensured that they were kept in a well-defined space which allows for the easy addition of strategies without the need for large additions to the Python code.

And, as of now, there has been little work done in a multi-agent simulation setting using these languages, particularly ProbLog. This presents an interesting and new horizon to explore.

5.2.3 ProbLog

Previous to this project, ProbLog had not been used in an agent-simulation setting, and part of the reason for using it here was to assess its suitability in this context. Our hypothesis for its suitability is that it is a declarative language that has the added bonus of probability that could be utilized in a probabilistic simulation setting.

5.2.3.1 Challenges with ProbLog

Originally, it was attempted to code the greedy and ConsideringOthers strategies in ProbLog utilising the inbuilt probabilistic aspect of the language. However, in the attempt to do this challenges were faced. Standard ProbLog does not have an inbuilt method of handling utilities or accessing the probabilities of the facts directly. It was therefore not possible to calculate the expected utility of the objects while simultaneously using ProbLog's inbuilt method of probability. For this reason, a custom implementation of probability is used for most strategies. This works due to the simple nature of the probabilistic calculations in the world setups.

A question arises here as to whether this was the best approach to take. It may seem counter-intuitive to be using an implementation of probability separate from the inbuilt probability element of the probabilistic language itself. It might be thought that it would have been better to use Prolog rather than ProbLog considering that the probabilistic element was not being used anyway. However, it was decided to stick with using the ProbLog compiler due to ProbLog's easily accessible Python API. DTProbLog was used for the ProbLog cell strategy, so keeping all of the programming in ProbLog and its extension DTProbLog kept the consistency higher than using both ProbLog and Prolog compilers.

Another issue that was faced when using ProbLog was infinite cycle errors. Initially, the attempt at implementing the ConsideringOthers strategy was to formulate simple rules that defined what the correct goal item should be. However, this caused an infinite cycle error since when searching for which item an agent would go for, this rule would have to be checked again and again forever.

To solve this, a list approach was used to keep track of items that were still an option for the agent to get. Items that no longer needed to be considered could be removed, meaning that there was no infinite cycle.

5.2.4 DTProbLog

DTProblog was used for one of the implemented strategies. In theory, DTProbLog is very suitable for the application of an agent simulation, it answers decision questions given to the program and replies with whether a decision should be taken or not. Utilities can be defined on certain predicates being true which is what the decision is based on.

5.2.4.1 Challenges with DTProbLog

However, it was found that DTProbLog is quite limiting in that unlike ProbLog it does not have queries. Due to this, using it for some of the strategies was quite difficult, and ProbLog was switched to instead.

5.3 Simulation

The Python package **simulation** houses the classes involved in the running of the simulation, including **World** and **WorldObject**.

5.3.1 A simulation time-step

In each time-step of the simulation, each agent performs its action. A consideration might be that this gives the agents that perform their actions earlier in the list an advantage. However, we investigate this idea in Section 6.3 and find there is little significant advantage.

An alternative would to be set our simulation up in a way that allowed the concurrent actions of all agents (much like in real life). However, due to us implementing a discrete time-step-based system, implementation of completely concurrent actions would not be possible as it may lead to agents ending up in the same position on the map, or attempting to eat the same item at the same time.

The events in a time step happen in the following order: agent's age is updated; agent's health is updated; check for death; eat item if can; otherwise, update goal item; move towards goal item.

5.4 World

A square 2D world is defined as the environment for the simulation to take place made up of cells. A world has a size which specifies the length of its sides. Within each cell of the world, there can only be one object (an agent or item).

The world was defined as a discrete 2D grid. It might be thought that a continuous environment would be more appropriate, but in a sense, the representation of the world map doesn't matter, as many of the strategies (the item-goal-based ones) could easily be carried over to a continuous world representation or 3-dimensional - or even n-dimensional - one. The item-based strategies could be used in a continuous world as the

specific method of moving towards the goal item is dependent on the Python simulation package.

The **WorldObject** class is used to represent any object in the world. This could be an agent or an item. **Agent** is a subclass of **WorldObject**. It holds information about the position, current goal item, and attributes such as current health.

5.5 Strategy

For each defined strategy, a corresponding instance of the Python Strategy class was created for it. Each strategy object has an embedded ProbLog or DTProbLog program.

The strategy program interfaces with the simulation Python program through query responses. At each time step, a strategy program is generated in ProbLog or DTProbLog for each agent (apart from when communication is on), adding the facts about its position in the world and its relation to other objects to the strategic rules which stay constant. The resultant program is queried to find what the next goal item or goal cell should be. Further details are given in Section 5.7.

5.5.1 One strategy per world

In our simulation set-up, a World is associated with a single strategy. It might be possible to enhance the system by enabling multiple strategies to be used in the same world, which could result in direct competition between agents with different strategies. It was decided not to allow this due to the ConsideringOthers strategy assumes that all agents in the simulation are following it. Thus, if agents with the greedy strategy were also present in the same world, the ConsideringOthers agents would perform suboptimal moves.

5.6 User interface

A graphical user interface (Figure 5.2) was created to easily allow the user to go through the time steps of the simulation, pick settings for the simulation, save and load set-ups, and see the simulation graphically. To represent the agents and objects in the world, the user is shown the world state at the time step that they are currently on. Pixels of different colours represent different types of objects, a colour key is dynamically generated depending on the specific types of objects in the set-up.

The user interface was built considering the choices which were important for the user to have when specifying the set-up of the simulation. Here we discuss the options which were given.

The **Next** button moves the simulation onto the next time step. When it is clicked, the world's Strategy program is constructed and queried, to find the next goal-item or goal-cell. **Restart** randomises a new simulation set-up and starts it from the beginning. **Save** saves the current simulation set-up, and **Load** loads a recent simulation set-up from a file. Loading a world restores all agents and objects to the world. Sliders

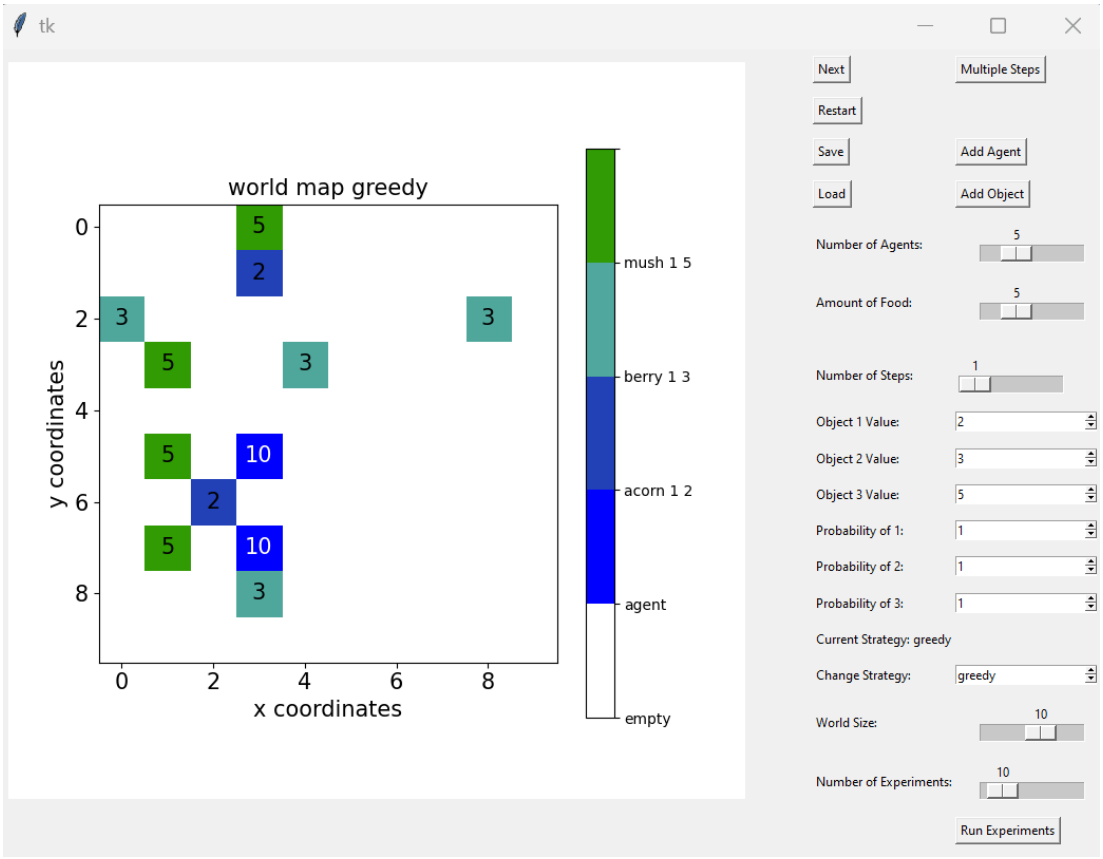

Figure 5.2: A screenshot taken of the user interface of the system. For a larger version of the user options, see A.1.

Number of agents and **Amount of food** can be used to select the amounts of each. When the simulation is restarted, the amounts specified by the sliders will be used.

The user has the option to change the value and probability of food from within the window using the Object Value and Probability number selectors. The user is also provided with the option to select an object regeneration value. This is a rate that describes the amount of items that reappear in the world as the simulation progresses. It is equal to the probability of an item appearing on the map at any time step.

The user can change the strategy of the world at any time step. This is a nice feature to have as it's interesting to see how the agents' behaviour changes immediately when changing the strategy.

The user can choose to run multiple time steps. This gives the user an easy way to see the simulation play out. The user also has the option to run experiments. The user can select how many experiments they'd like to run; the experiments will then be carried out and the results saved to files in CSV form. This data can then be plotted through a spreadsheet or other means. This is the technique that was used when performing the Experiments in Section 6.

5.7 ProbLog Maker

A module, **problog_maker**, was written to dynamically generate ProbLog programs suited to the world's setup. This module is invoked at every time step to produce appropriate ProbLog programs, guiding the determination of the target goal item or cell.

This technique was decided upon due to the changes that are made to the world in each time-step causing the ProbLog model to need updating.

Each strategy has a corresponding rules text file that is static. Depending on the world set-up background knowledge is then added to the program string. The background knowledge is generated to declare the agents, and other objects in the world. As well as this, object positions, probabilities, and values are set.

A potential downside of using the ProbLogMaker is that it might slow down the simulation as a whole due to many ProbLog programs needing to be produced through string manipulation. In some techniques such as ConsideringOthers, a ProbLog program was needed to be produced for every agent in the world at every time step. The aforementioned collaboration setting (Section 4.1) goes some way to alleviating the lag.

5.8 Limitations

One significant limitation of the ABM software is the speed of certain strategies, particularly the ConsideringOthers strategy. Although efforts were made to optimize performance, such as implementing the communication option, further enhancements are needed to ensure efficient execution. For example, separating the simulation process from the user interface on different threads could prevent the software from becoming

unresponsive during simulation steps. Additionally, optimizing the strategy computations to reduce redundant recalculations of the goal item at every move could improve overall efficiency.

As was discussed earlier, due to the complexity of the problem that was investigated, the probabilistic element of ProbLog was not as deeply utilised as was initially hoped when setting out on the project.

Chapter 6

Results and Analysis

In this section, we investigate the performance of the created strategy-programs across multiple evaluation metrics. Drawing from the diverse range of goals outlined in the preceding section 3.2, we assess how these strategies fare in achieving different objectives. The ProbLog cell strategy is excluded from the experiments due to it being significantly slower. Our focus remains on analyzing the four top-performing strategies.

The metrics considered in the following experiments are:

- total number of items consumed in an agent's lifetime
- total amount of value consumed in an agent's lifetime
- the number of items present in a world over time

In these experiments, it was decided to use three item types with a spread of item values of 2, 3 and 5. Including a range of item values was important so as to compare strategies that act depending on value with those that act on distance. The values were kept not too widely spread out, as high values could potentially skew the average values for the total value consumed.

The size of the world chosen for these experiments was 10. It was found that this size has a good balance of speed and space for the agents to move in. Each of the experiments were run for exactly 10 time steps. It was found that in experiments with object regeneration off, usually all items had been consumed (other than in the Greedy-No-Distance strategy, we will discuss this shortly) within 10 moves.

6.1 Experiment 1: The effect of strategy on the average number of food consumed in an agent's lifetime

This experiment was repeated twice with object regeneration set to 0 (which means it's switched off), and with it set to 0.2. In both instances, world simulations were run 30 times for each strategy.

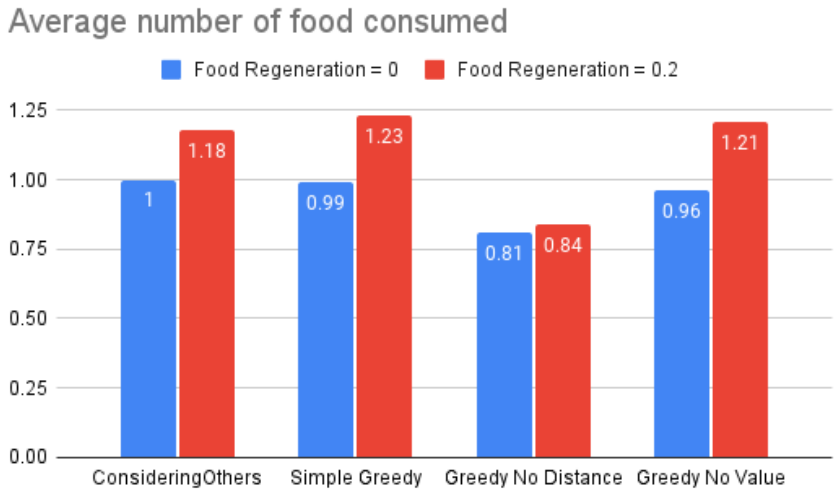

Figure 6.1: A graph showing the average number of items consumed per lifetime, per strategy. The graph displays data for both versions of the experiment, with object regeneration on (blue), and off (red)

The hypothesis for this experiment was that the ConsideringOthers strategy would perform better on this metric due to its more advanced reasoning.

6.1.1 Object regeneration = 0

Figure 6.1 shows the results of the average number of items consumed in the simulation run per agent. The results of this experiment are shown in blue. As hypothesised, the ConsideringOthers strategy performs best on this metric. For the greedy strategies, as we are just looking at the number of food eaten, it makes sense that greedy_no_value is the best at this. It does not put more value on foods with higher value and instead only prioritises objects which are closer in distance. Therefore they are able to consume a higher number of objects, even if they were low in value.

6.1.2 Object Regeneration = 0.2

The experiment was repeated but with food regeneration on in order to see if the results matched for when this setting was on as well as to see if there was any change introduced into the results due to the object regeneration setting being on.

The results for the experiment match the results of the previous, as expected, with the number of items being consumed by agents employing the Greedy-No-Value being higher and the number of food eaten by Greedy-No-Distance being lower. The results can be seen in Figure 6.1 in red along with the results of the experiment without regeneration in blue.

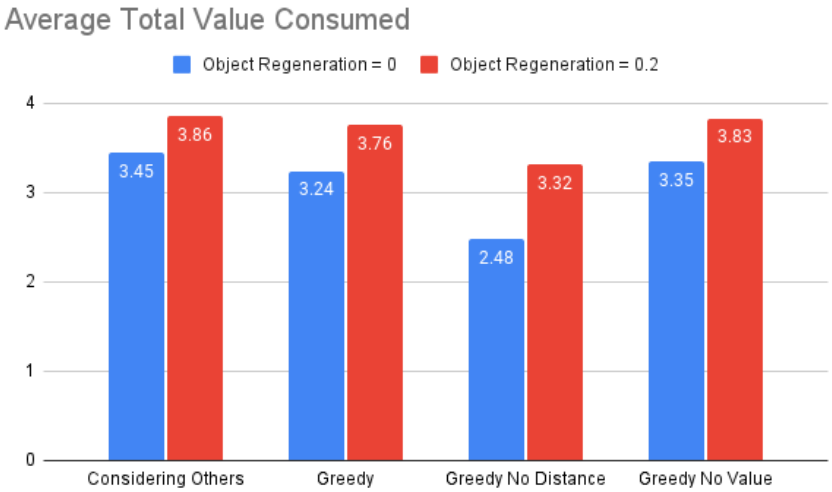

Figure 6.2: A graph showing the average value consumed per lifetime, per strategy. The graph displays data for both versions of the experiment, with object regeneration on (blue), and off (red)

6.2 Experiment 2: Effect of strategy on total value consumed

In Experiment 2, the effect of strategy on the total average value consumed in an agent's lifetime was studied. The experiment setup included 5 agents and 5 items. The total value consumed in an agent's lifetime is a key metric and it was an idea that the ConsideringOthers strategy was built around. It was hypothesised that the ConsideringOthers strategy would perform best, due to its more advanced reasoning for which items are worth pursuing. It was expected that the Simple Greedy strategy would perform second best, followed by the Greedy-No-Distance and Greedy-No-Value strategies. It was uncertain which of the last two would perform better.

The results of the experiment can be seen in Figure 6.2. As hypothesised, the ConsideringOthers Strategy performs best on this metric, with agents consuming 3.45 value in a lifetime on average.

Greedy-No-Distance performed worst, as in the previous experiments. This is likely due to the same reasons of how they are more likely to go large distances, ignoring options of items which though worse than the best, would provide them with worthwhile value.

Simple-Greedy and Greedy-No-Value appear to perform about the same on this metric. This is perhaps due to it being more likely for a Simple-Greedy agent to get food that is the closest to it than some slightly further away that *would* be better if they got it. These agents are less likely to waste as much time moving towards food than they do not consume in the end.

To check if there was any difference in the result as well as to validate the results, the experiment was repeated but with the object regeneration rate set to 0.2. The results followed the same pattern but with slightly higher averages due to the increased food

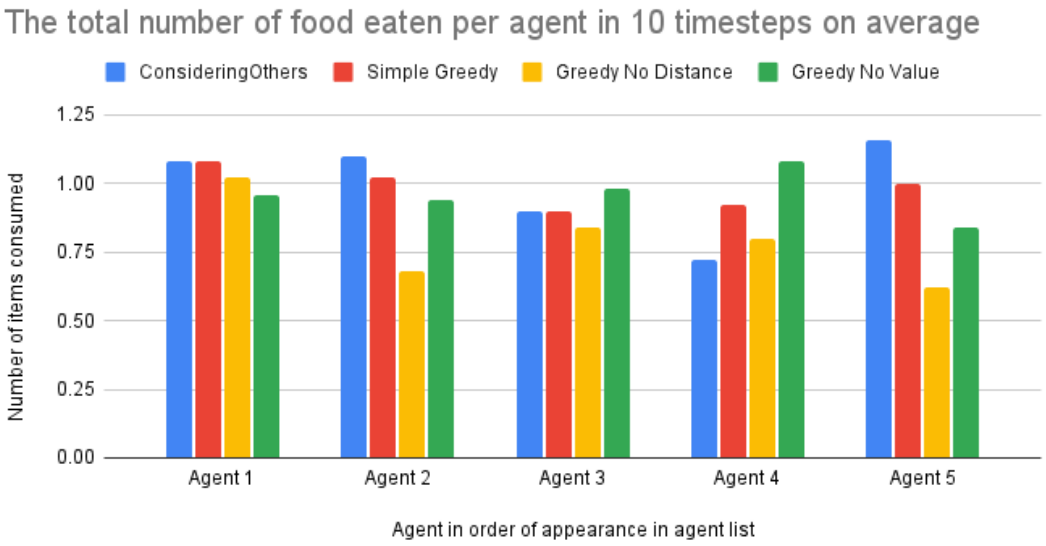

Figure 6.3: A graph showing the average total number of items eaten per agent. For each strategy, the experiment was run 50 times.

amounts in the world.

6.3 Experiment 3: The effect of position in the agent list on the quantity of food eaten in an agent's lifetime

The hypothesis for this experiment was that agents further up in the list of agents will have an advantage due to them being able to reach items before other agents in the instance that they begin in positions equidistant from an item.

To evaluate the truth of this we initially looked at data from 25 experiment runs. After the first 25 runs, it was unclear whether there was any significant pattern in the results. There was a question as to whether the results were statistically significant. For this reason, the experiment was repeated another 25 times and those results were plotted too. The combined results of the 50 experiments are shown in Figure 6.3.

Initially from this graph it appears as though the effect of being an agent higher up in the list has little to no effect. This is since the results show no clear evidence of there being more food consumed for agents higher up in the list. However, averaging the data amongst all strategies provides clearer insights as can be seen in Figure 6.4. This shows that there may be some slight advantage in being higher up in the list, particularly for being agent 1. However, this result will be made particularly prominent due to the low number of timesteps investigated. It is likely that had further time-steps been used in the experiment, this result would be reduced.

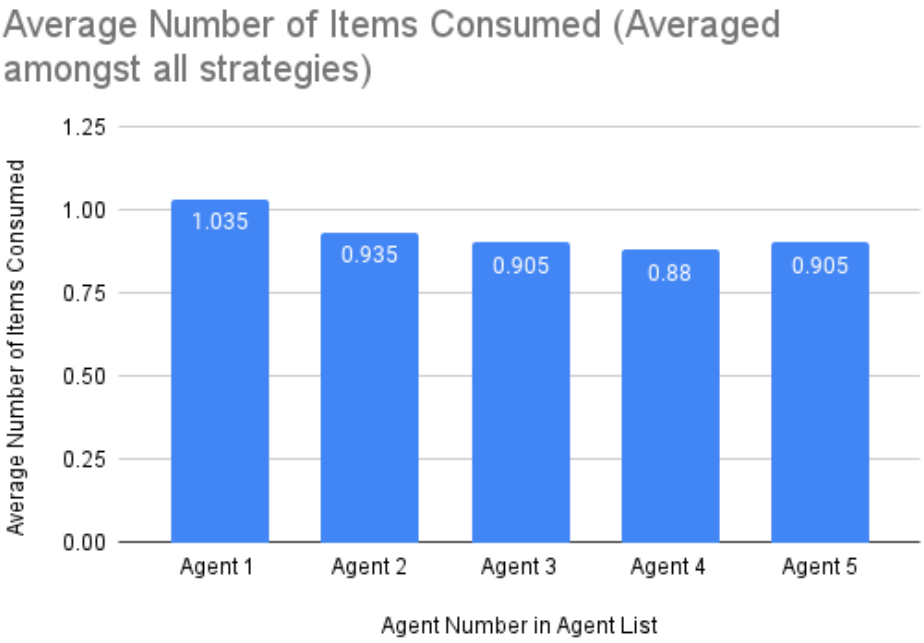

Figure 6.4: A graph depicting the average number of items consumed depending on the agent's position in the agent list, averaged over 200 experiments (50 for each strategy).

6.4 Experiment 4: The effect of strategy on the trend of food eaten over time

The hypothesis for this experiment was that the ConsideringOthers strategy would perform best due to less time being spent pursuing items that the agent wouldn't end up getting, as is often the case in the greedy strategies. It was also hypothesised that the Greedy-No-Distance strategy would perform worst, due to spending more of their time moving towards valuable items and ignoring low-value items.

This experiment aimed to see the effect of strategy on how the amount of food eaten changes. This particular experiment was performed with 10 agents and 10 items. object regeneration was off. Simulations for all strategies were run 30 times each.

Figure 6.5 shows the results of the experiment. It can be seen that on average food in worlds employing the ConsideringOthers strategy are consumed in fewer timesteps than in the Greedy strategy world.

The results of this experiment make sense. The reason that Greedy-No-Distance items are removed from the world on average at a significantly lower rate is due to the fact that in that strategy all agents will go towards only the most valuable item at the same time, meaning most agents waste their movements pursuing items they don't end up getting. ConsideringOthers, Greedy, and Greedy-No-Distance have similar performance on this metric, with ConsideringOthers seeming to perform slightly better than at least the Simple Greedy strategy. This would make sense due to the increased reasoning that comes with the ConsideringOthers strategy compared to the Simple Greedy strategy,

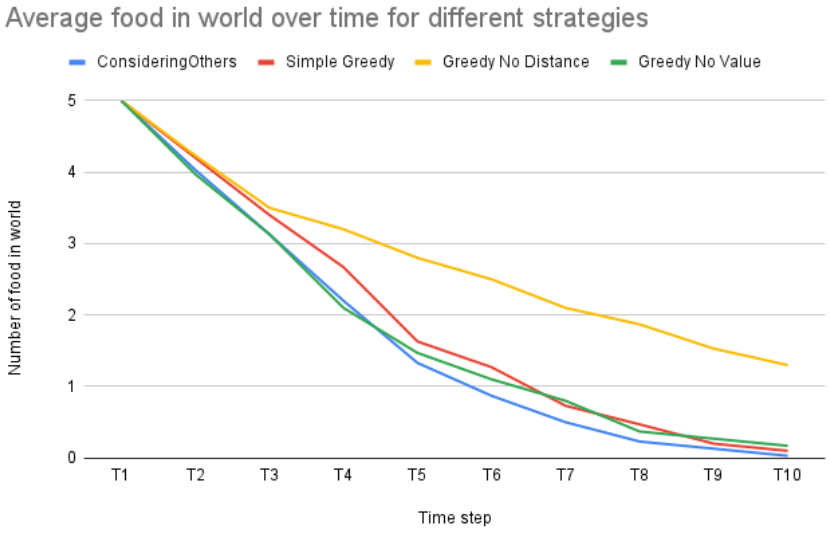

Figure 6.5: A graph plotted of the average food over time for each of the four best-performing strategies over 10 timesteps. The graph includes 30 runs of the experiment for each strategy.

leading to less wasted movements.

6.5 Interpretation of Results

In this section, we discuss the Experiment Results discuss what they suggest about which strategies are suited for different specific scenarios.

An observation would be that the strategies appear to perform generally equally in comparison with each other regardless of metric - as in whether their success is measured by the number of items consumed or the total value of items consumed. This might be expected where all, or nearly all, food items are consumed in the experiment-run as the total value consumed will be the same either way.

An exception to this is seen in the comparison between results for Experiment 1 and Experiment 2 in the Greedy-No-Distance strategy. It appears that for the metric of the number of items consumed, there appears to be little difference when regeneration is on or off. However, when regeneration is on it does significantly better when the metric is value.

This result can be explained by the fact that these agents all move towards the most valuable item available in the world at all times. Without object regeneration on it is likely that high-value items will all be consumed, leaving only the low-value items remaining. However, the setting of object regeneration on allows more items into the world, therefore increasing the probability of there being a higher number of high-value objects. This is amplified by the inefficiency of the strategy which means a smaller proportion of items are consumed overall, but with regeneration on, those that *are* eaten will have higher value on average.

In general, the ConsideringOthers strategy appears to perform better in the majority of the experiments. In Experiment 1, which studies the number of items consumed, it performs marginally worse then Simple-Greedy and Greedy-No-Value. This makes sense, as the strategy does not prioritise the number of items eaten and instead prioritises gaining the most value. In Experiment 2, where the metric is total value consumed, the ConsideringOthers strategy performs best, as expected.

It can perhaps be seen as encouraging that the sophisticated logical approach appears to be the most efficient.

Chapter 7

Conclusions

7.1 Overview

The Agent-Based Modelling Software was successfully produced, providing the user with the key features which were originally proposed, which were to specify the number of agents and items. Agents and items are displayed on a grid where time-steps can be looked through, setups of the world can be specified including numbers of agents and items, and the size of the world and the strategy it employs. Five strategy programs were created in ProbLog that can be run through the Python ABM Software, each incrementally improving on the last. These strategies were evaluated on several metrics.

7.2 Future Work

Future iterations of the software could include features like adding a limited field of vision to agents, which could expedite strategy computations and introduce interesting dynamics. Another potential enhancement is the implementation of agent consumption, similar to Predator-Prey simulations. While this would add complexity to strategy design, it could lead to more nuanced and realistic agent behaviours.

Graphical visualization of simulation data is another area for expansion. Integrating tools like Python library MatPlotLib to generate graphs of metrics such as food distribution over time within the software could provide easily accessible valuable insights into simulation dynamics.

Topics such as theory of mind and machine learning, which were not pursued in this project, could be explored opening avenues for future research and development.

The strategies that were implemented showed some effectiveness, although there is clear room for more advanced strategies to be implemented, as was discussed in Section 4.5. The effective development of agent strategies in ProbLog and other declarative languages continues to be an interesting field with many paths to go down.

Bibliography

[1] Abhishek Balasubramaniam and Sudeep Pasricha. Object detection in autonomous vehicles: Status and open challenges, 2022.

[2] Luuk Bom, Ruud Henken, and Marco Wiering. Reinforcement learning to train ms. pac-man using higher-order action-relative inputs. In *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 156–163, 2013.

[3] F. Castiglione. Agent based modeling. *Scholarpedia*, 1(10):1562, 2006. revision #123888.

[4] Prashant Doshi, Piotr Gmytrasiewicz, and Edmund Durfee. Recursively modeling other agents for decision making: A research perspective. *Artificial Intelligence*, 279:103202, 2020.

[5] Frédérick Garcia and Emmanuel Rachelson. Markov decision processes. *Markov Decision Processes in Artificial Intelligence*, pages 1–38, 2013.

[6] David Hall, Feras Dayoub, John Skinner, Haoyang Zhang, Dimity Miller, Peter Corke, Gustavo Carneiro, Anelia Angelova, and Niko Sünderhauf. Probabilistic object detection: Definition and evaluation. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, pages 1031–1040, 2020.

[7] IRD. Gama platform, 2020-2023.

[8] Angelika Kimmig, Bart Demoen, Luc De Raedt, Vítor Santos Costa, and Ricardo Rocha. On the implementation of the probabilistic logic programming language problog. *CoRR*, abs/1006.4442, 2010.

[9] Fredrik Lundh. An introduction to tkinter. *URL: www. pythonware. com/library/tkinter/introduction/index. htm*, 1999.

[10] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *nature*, 518(7540):529–533, 2015.

[11] RedfishGroup Owen Densmore. Agentscript, 2012-2023.

[12] David V Pynadath and Stacy C Marsella. Psychsim: Modeling theory of mind with decision-theoretic agents. In *IJCAI*, volume 5, pages 1181–1186, 2005.

[13] Anand S. Rao and Michael P. Georgeff. Modeling rational agents within a BDI-architecture. In James Allen, Richard Fikes, and Erik Sandewall, editors, *Proceedings of the 2nd International Conference on Principles of Knowledge Representation and Reasoning*, pages 473–484. Morgan Kaufmann publishers Inc.: San Mateo, CA, USA, 1991.

[14] Taisuke Sato. A statistical learning method for logic programs with distribution semantics. In *International Conference on Logic Programming*, 1995.

[15] Taisuke Sato and Yoshitaka Kameya. Generative modeling with failure in prism. pages 847–852, 01 2005.

[16] Thon I. van Otterlo M. De Raedt L. Van den Broeck, G. Dtproblog: A decision-theoretic probabilistic prolog. 24.

[17] Joost Vennekens, Marc Denecker, and Maurice Bruynooghe. Cp-logic: A language of causal probabilistic events and its relation to logic programming. *Theory and Practice of Logic Programming*, 9(3):245–308, 2009.

[18] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

Appendix A

First Appendix

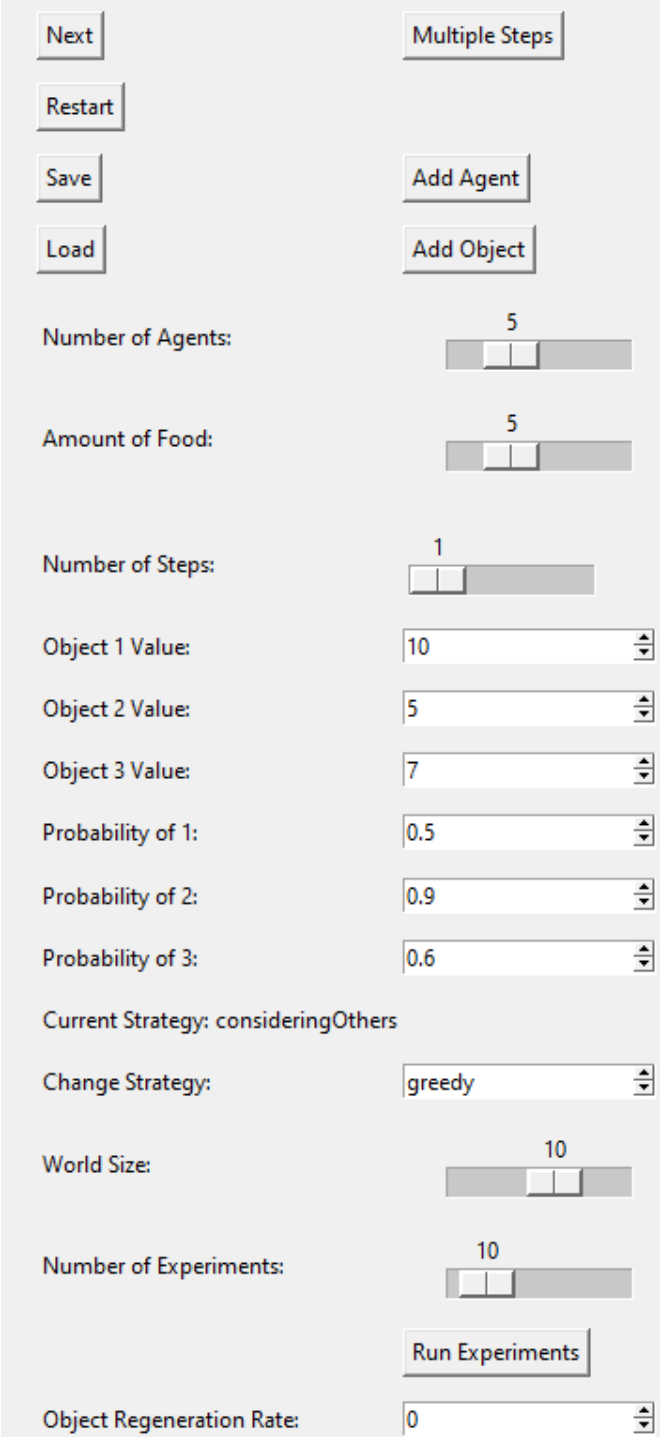

Figure A.1: An elarged image of the options that the ABM Software allows the user.